Layering in Provenance Systems

Layering in Provenance Systems

Kiran-Kumar Muniswamy-Reddy, Uri Braun, David A. Holland,
Peter Macko, Diana Maclean, Daniel Margo, Margo Seltzer, Robin Smogor
Harvard School of Engineering and Applied Sciences
pass@eecs.harvard.edu

Abstract

Digital provenance describes the ancestry or history of a digital object. Most existing provenance systems, however, operate at only one level of abstraction: the system call layer, a workflow specification, or the high-level constructs of a particular application. The provenance collectable in each of these layers is different, and all of it can be important. Single-layer systems fail to account for the different levels of abstraction at which users need to reason about their data and processes. These systems cannot integrate data provenance across layers and cannot answer questions that require an integrated view of the provenance.

We have designed a provenance collection structure facilitating the integration of provenance across multiple levels of abstraction, including a workflow engine, a web browser, and an initial runtime Python provenance tracking wrapper. We layer these components atop provenance-aware network storage (NFS) that builds upon a Provenance-Aware Storage System (PASS). We discuss the challenges of building systems that integrate provenance across multiple layers of abstraction, present how we augmented systems in each layer to integrate provenance, and present use cases that demonstrate how provenance spanning multiple layers provides functionality not available in existing systems. Our evaluation shows that the overheads imposed by layering provenance systems are reasonable.

1 Introduction

In digital systems, *provenance* is the record of the creation and modification of an object. Provenance provides answers to questions such as: *How does the ancestry of two objects differ? Are there source code files tainted by proprietary software? How was this object created?* Most existing provenance systems operate at a single level of abstraction at which they identify and record provenance. Application-level systems, such as Trio [29], record provenance at the semantic level of the application – tuples for a database system. Other application-level solutions record provenance at the level of business objects, lines of source code, or other units with semantic meaning to the application. Service-oriented workflow (SOA) approaches [8, 9, 23], typically associated with workflow engines, record provenance at the level of workflow stages and data or message exchanges. System-call-based systems such as ES3 [3], TREC [28], and PASS [21] operate at the level communicated via system calls – processes and files. In all of these cases, provenance increases the value of the data it describes.

While the provenance collected at each level of abstraction is useful in its own right, integration across these layers is crucial but currently absent. Without a unified provenance infrastructure, individual components produce islands of provenance with no way to relate provenance from one layer to another. The most valuable provenance is that which is collected at the layer that provides user-meaningful names. If users reason in terms of file names, then a system such as PASS that operates at the file system level is appropriate. If users want to reason about abstract datasets manipulated by a workflow, then a workflow engine's provenance is appropriate. As layers interoperate, the layers that name objects produce provenance, transmitting it to other layers and forming relationships with the objects at those different layers. For example, this might associate many files that comprise a data set with the single object representing the data set. PASS captures provenance transparently without application modification, but it might not capture an object's semantics. If applications encapsulate that semantic knowledge, then those applications require modification to transmit that knowledge to PASS. In summary, application and system provenance provide different benefits; the value of the union of this provenance is greater than the sum of its parts.

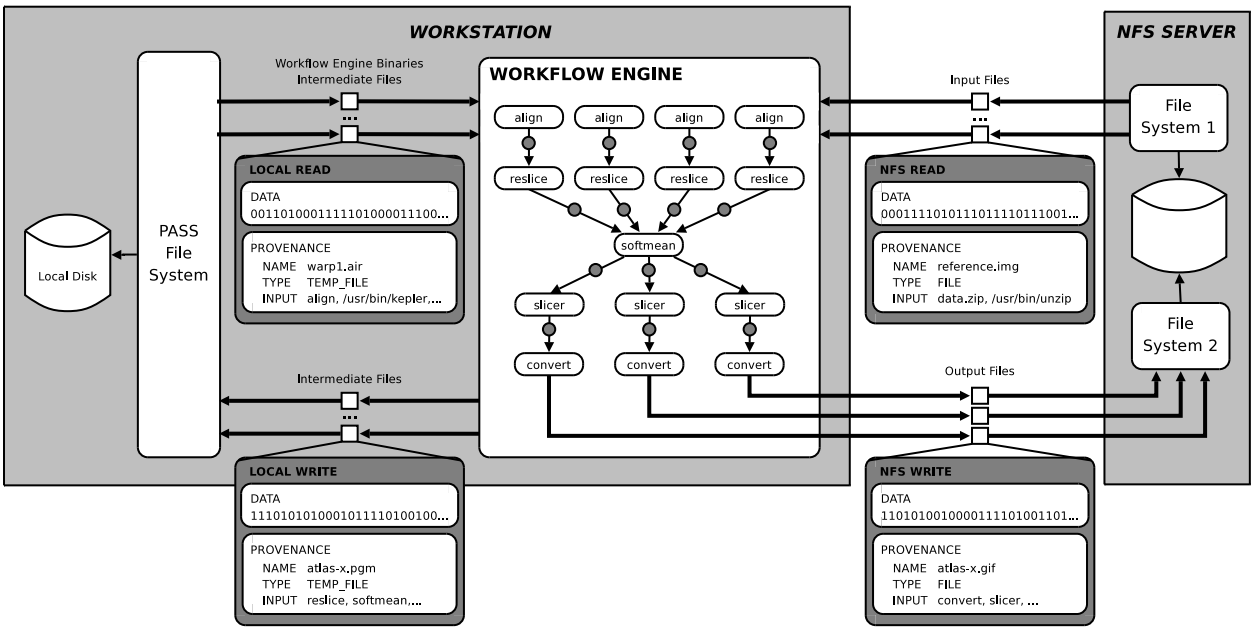

Figure 1: **Example of a layered architecture introduced in Section 1.** This scenario demonstrates a workflow application running on a workstation, but accessing its inputs and outputs on remote file servers. The workflow engine, local file system, and remote file system all capture provenance, but only by integrating that provenance can we respond to queries that require complete ancestry of an object.

Consider the scenario in Figure 1. A workflow engine running on a workstation reads input files from an NFS-mounted file system, runs a workflow that produces intermediate files stored on the the workstation's disk, and ultimately produces three output files that are stored on a second NFS-mounted file system. Imagine that we run the workflow on Monday, and unbeknownst to us, a colleague modifies one of the input files on Tuesday. When we run the workflow on Wednesday, we find that it produces different results. If we capture provenance only on the local or remote file systems, we cannot see any of the processing stages that occur inside the workflow engine. If we capture provenance inside the workflow engine, then we lose track of the fact that one of the inputs to the workflow changed, since it changed in a manner transparent to the workflow engine (on a remote server). Properly tracing the ancestry of the output file requires the full provenance chain: from one remote server through the local system and workflow engine to another remote server.

This example illustrates the challenges that we encountered in developing a layered provenance architecture. We will use this example throughout the rest of the paper to discuss these challenges and describe our solutions to them. We describe the PASSv2 system, a new version of PASS that enables the seamless integration of provenance from different layers of abstraction. In such a layered system, data and provenance flow together through the different layers. The resulting system provides a unified provenance infrastructure allowing users to get answers to queries that span layers of abstraction, including information about objects in different applications, the local file system, remote file systems, and even those downloaded from the Web. Furthermore, the system allows an arbitrary number of layers to stack on each other (we demonstrate a three layer stack in Section 7).

The contributions of this work are:

- Use cases demonstrating the utility of integrating provenance-aware applications and a provenance-aware operating system.
- An architecture for provenance-aware systems that integrates provenance across multiple abstraction layers.
- A prototype demonstrating the capabilities of this layered architecture through the creation of an integrated set of provenance-aware components: NFS, a workflow engine, a browser (`links`), and an initial runtime Python provenance tracking wrapper.

The rest of this paper is organized as follows: In Section 2, we place this work in the context of existing systems and introduce the idea of integrating provenance across semantic layers in a system. In Section 3, we present use cases that highlight the benefits

of layering. In Section 4, we discuss the fundamental challenges inherent in building systems that integrate provenance across multiple layers, and in Section 5, we present our layered architecture, demonstrating how it addresses these challenges. In Section 6, we describe the provenance-aware applications and remote file server we developed. Section 7 presents the cost of providing these features in terms of time and space overheads. In Section 8, we discuss related work, and we conclude in Section 9.

2 Background

Previous work exists at each of the layers discussed in the introduction, but there is no approach that integrates across the different levels of abstraction to provide a unified solution.

At the domain-specific level, systems like GenePattern [10] provide provenance for environments in which biologists perform routine analyses. Experiments done within the analysis environment record and maintain information about the particular algorithms and parameters used, but this information can be lost if data is manipulated outside of the environment.

Tracking provenance at the level provided by workflow engines – such as Pasoa [23], Chimera [9], and Kepler [2] – allows users to group collection of related objects into single logical entities. For example, scientists frequently refer to logical data sets containing hundreds or thousands of individual files. These systems can answer queries such as: *What were all the output files of a particular experiment?* or *What version of the software release are we using for this analysis?* These systems lose some of the semantic knowledge available at the domain-specific level, but do provide the ability to handle abstractions such as data sets.

System level solutions like ES3 [3] and PASSv1 [21] capture information at the operating system level, losing both the semantic information of domain-specific solutions and also the relationships among data sets and processing units found in workflow engines. However, these systems provide a wealth of information about the environment in which objects are created, such as the specific binaries, libraries, and kernel modules in use.

Provenance is not the same as a security label, and provenance systems are different from label-based security systems such as HiStar [31], Asbestos [6], and Flume [18]. These systems track information flow but not in sufficient detail for general provenance querying. ES3 and PASSv1 capture not only the *fact* of relationships among data sets but also the *means*, including data such as process arguments and environment variables, and support queries over this information.

All of these solutions fundamentally fail to account for the different levels of abstraction at which users need to reason about their data and processes. Users should be able to work at any or all levels as desired, rather than being limited to one. This requires being able to relate objects that appear in one layer to their manifestations in other layers.

It seems that perhaps it is sufficient for each system to generate its own provenance independently and then use the names of objects to link the layers together, as by a relational join. However, the Second Provenance Challenge [25] showed that even at a single level of abstraction, uniform object naming is both fundamental to provenance interoperability and nontrivial. Across abstraction boundaries, it is harder yet. For example, an object that exists in one layer may in other layers have some local manifestation, such as a collection of files forming a data set, but no clearly defined name. Thus, using only object names or identifiers is not sufficient.

Providing a uniform basis for object identity and linkage requires an integrated provenance solution with explicit layer support, where data moving from one layer to the next carries its provenance with it.

3 Use Cases

We have explored provenance collection in a variety of different contexts ranging from NFS to web browsers. We began with NFS, as a large number of users store their data on network-attached storage. Developing provenance-aware NFS (PA-NFS) helped us understand how to extend provenance outside a single machine. Next, we decided to explore integrating a provenance-aware workflow engine with PASSv2 as most prior provenance systems operated at the level of a workflow engine [2, 9, 23]. We selected the Kepler open-source workflow enactment engine [2] in which to explore integrating workflow provenance with PASSv2. We then explored adding provenance collection to an application that bears little similarity to workflow engines and operating systems: a web browser. We used the *links* text based web browser for this purpose. Last, we built a set of Python wrappers to capture provenance for Python applications.

In this section, we present scenarios and differentiate the problems that provenance-aware systems can address with and without layering.

3.1 Provenance Aware NFS

Use Case: Finding the Source of Anomalies

Scenario: Implementing the scenario depicted in Figure 1, we use Kepler to execute the Provenance Challenge

workflow [24], reading inputs from one NFS file server and writing outputs to another. Between two executions, unbeknownst to us someone modifies an input file. When we examine the new output, we see that it is different from the old output, and we would like to understand why.

Without Layering: If we examine only the Kepler provenance, we would think that the two executions were identical, since the change in the input file is invisible to Kepler. If we examine only the PASSv2 provenance, we would see that there was a different input to the Kepler workflow, but we would not know for sure that the input was actually used to produce the output since we cannot see how the multiple inputs and outputs are related.

With Layering: If Kepler runs on PASSv2, then the PASSv2 provenance store contains the provenance from both systems. Hence, it is possible to both determine and verify that the input file modification was responsible for the different output.

3.2 Provenance-Aware *links*

Use Case: Attribution

Scenario: A professor is preparing a presentation and has a number of graphs and quotes that have been previously downloaded from the Web. She copies these objects into the directory containing the presentation. Now, she would like to include proper attribution for them, but none remain in the browser's history and some of them are no longer even accessible on the Web.

Without Layering: Any browser can record the URL and name of a downloaded file and, when the site is revisited, can verify if the file has changed. (In fact, this is how most browser caches function.) However, if the user moves, renames, or copies the file, the browser loses the connection between the file and its provenance. The provenance collected by PASSv2 alone is insufficient as it only records the fact that the file was downloaded by the browser.

With Layering: A provenance-aware browser generates provenance records that include the URL of the downloaded file and transmits them to PASSv2 when it writes the file to disk. PASSv2 ensures that the file and its provenance stay connected even if it was renamed or copied. Our absent-minded professor can now determine the browser provenance for the file included in the presentation.

Use Case: Determining Malware Source

Scenario: Suppose Alice downloaded a codec from a web site. Suppose further that Eve, unbeknownst to Alice, has hacked the web site and caused this codec to contain malware. Alice later discovers that her computer has been infected. She would like to be able to find the origin of the malware and the extent of the damage.

Without Layering: Alice can traverse the provenance graph recorded by PASSv2 (similar to Backtracker [17] and Taser [11]) to identify and remove the malware binaries and recover any corrupted files. However, PASSv2 by itself cannot identify the web site from which the malware was downloaded. Conversely, a provenance-aware browser can identify the web site from which a known malware file was downloaded, but it cannot track the spread of that malware through the file system.

With Layering: A provenance-aware browser integrated with PASSv2 can help identify the web site that Alice was visiting when malware was downloaded, any linked third party site where the malware download originated, as well as other details about the browsing session (for example, the user may have been redirected from a trusted site). It can also find other files and descendants of files downloaded from the same web site, which may now be suspect. Layering with PASSv2 also provides an extra level of protection. The malware can compromise the browser, but to hide the fact that it was ever on the system, it also needs to compromise the operating system. If the operating system is compromised, we can ensure the integrity of the provenance collected before the compromise by using a selective versioning secure disk system [27].

3.3 Provenance-Aware Python

Use Case: Determining Data Origin

Scenario: Through approximately 400 experiments on 60 specimens over the course of a week, colleagues in Iowa State's Thermography Research Group developed a set of data quantitatively relating crack heating to the vibrational stresses on the crack. The experiment logs for these data were stored in a series of XML files by the team's data acquisition system. A team member developed a Python script to plot crack heating as a function of crack length for two different classifications of vibrational stress. Our goal was to determine the sources of the specific XML data files that contributed to each plot.

Without Layering: This might have been a simple problem for PASSv2, except that the analysis program reads in *all* the XML data files to determine which ones to use. PASSv2 reports that the plot derives from all the XML files. Provenance-Aware Python knows which XML documents were actually used, but it does not know the source files of those documents.

With Layering: In a layered Provenance-Aware Python/PASSv2 system, queries over the provenance of the resulting plot can report both the precise XML documents, the files from which they came, and the prove-

nance of those files.

Use Case: Process Validation

Scenario: They upgraded the Python libraries on one of their analysis machines, introducing bugs in a calculation routine used to estimate crack heating temperatures. The group discovered this bug after running the experiments and wanted to identify the results that were affected by the erroneous routine.
Without Layering: PASSv2 can distinguish which output files were generated using the new Python library, but cannot determine which of those files were generated by invoking the erroneous routine. Provenance-Aware Python can determine which files were generated by invoking the calculation routine, but cannot tell which version of Python library was used.
With Layering: Integrating the provenance collected by PA-Python and PASS identifies the files that have incorrect data, because they descend from both the new Python library and the calculation routine.

4 Challenges in Layering

Mapping objects between the different layers of abstraction is only one of the challenges facing layered provenance systems. We identified six fundamental challenges for a layered provenance architecture:

Interfacing Between Provenance-Aware Systems: The manner in which different provenance-aware systems stack is not fixed. A workflow engine might invoke a provenance-aware Python program (see Section 6.4) in one instance and in another instance be invoked by it. Thus, provenance-aware components must be able to both accept and generate messages that transmit provenance. We designed a single universal API appropriate for communication among PASSv2 components and also among different provenance systems. It took several iterations to develop an API that was both general and simple; we discuss the resulting Disclosed Provenance API (DPAPI) in Section 5.2.

Object Identity: As mentioned earlier, objects may be tangible at one layer and invisible at another. Imagine tracking provenance in a browser, as in Section 6.3. It would be useful to track each browser session as an independent entity. However, browser sessions do not exist as objects in the file system, so it is not obvious how to express a dependence between a browser page and a file downloaded from it. We show how the DPAPI makes objects from one layer visible to other layers and how the *distributor* lets us manage objects that are not manifest at a particular layer in Section 5.5.

Consistency: Provenance is a form of metadata; we need to define and enforce consistency semantics between the data and its metadata, so users can make appropriate use of it. The DPAPI bundles data and provenance together to achieve this consistency and our layered file system, Lasagna (Section 5.6), maintains this consistency on disk.

Cycles: In earlier work, we discussed the challenge of detecting and removing cycles in PASSv1 [21]. This problem becomes even more complicated in a layered environment. Since there are objects that appear at one layer and not at others, we may need to create relationships between objects that exist in different layers, and then detect and remove cycles that these relationships introduce. In Section 5.4, we discuss how the *analyzer* performs cross-layer cycle detection.

Query: Collecting provenance is not particularly valuable if we cannot make it available to a user or administrator in a useful fashion. We shadowed several computational science users to understand what types of queries they might ask a provenance system. After struggling through three generations of query languages for provenance, we incorporated the input from our users and derived the following list of requirements for a provenance query language [16]:

- The basic model should be *paths through graphs*;
- Paths should be first-class language level objects;
- Path matching should be by regular expressions over graph edges; and
- The language needs sub-queries and aggregation.

Query languages for semi-structured data proved the best match; our query language PQL (Path Query Language) derives from one of these. Section 5.7 provides a brief overview.

Security: While there has been research showing the use of provenance for auditing and enhancing security [13], there has been little work on security controls for the provenance itself. The fundamental provenance security problem is that provenance and the data it describes do not necessarily share the same access control. There is no universally correct rule that dictates which of the two (data or provenance) requires stronger control. For example, consider a report generated by aggregating the health information of patients suffering a certain ailment. While the report (the data) can be accessible to the public, the files that were used to generate the report (the provenance) must not be. The provenance must be more tightly controlled than the data. Conversely, a document produced by a government panel (the data) might be classified, but the membership of the committee and the identities of all participants in briefings (the provenance) may nonetheless be entirely public. The data carries stronger access control than the provenance. Creating an access control model for provenance is outside the scope of this

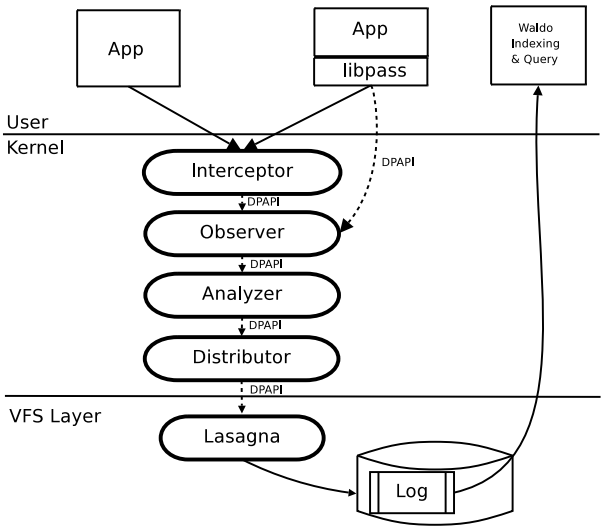

Figure 2: PASSv2 Architecture

paper; however, a related paper presents an in-depth discussion of the problem and our approach to solving it [4].

5 Architecture

We begin with a high level overview that introduces the main components of the PASSv2 system. Then we explain each component of the system in detail and show how it addresses the challenges discussed above.

5.1 Overview

From a user perspective, PASSv2 is an operating system that collects provenance invisibly. Application developers can also use it to develop provenance-aware applications. Figure 2 shows the seven main components of the PASSv2 system. These are:

libpass: `libpass` is a library that exports the DPAPI to user-level. Application developers develop provenance-aware applications by augmenting their code to collect provenance and then issuing DPAPI calls to `libpass`.

Interceptor: The interceptor intercepts system calls and passes information to the observer.

Observer: The observer translates system call events to *provenance records*. For example, when a process P reads a file A, the observer generates a record $P \rightarrow A$, indicating that P depends on A. Hence, together the observer and the interceptor generate provenance.

Analyzer: The analyzer processes the stream of provenance records and eliminates duplicates and ensures that cyclic dependencies do not arise.

Distributor: The distributor caches provenance for objects that are not persistent from the kernel's perspective, such as pipes, processes and application-specific objects (e.g., a browser session or data set) until they need to be materialized on disk.

Lasagna: Lasagna is the provenance-aware file system that stores provenance records along with the data. Internally, it writes the provenance to a log. The log format ensures consistency between the provenance and data appearing on disk.

Waldo: Waldo is a user-level daemon that reads provenance records from the log and stores them in a database. Waldo is also responsible for accessing the database on behalf of the query engine.

5.2 Disclosed Provenance API (DPAPI)

The DPAPI is the central API inside PASSv2. It allows transfer of provenance both among the components of the system and between layers. Applications use the DPAPI to send ("disclose") provenance to the kernel. The same interface is used to send provenance to the file system. The DPAPI consists of six calls: `pass_read`, `pass_write`, `pass_freeze`, `pass_mkobj`, `pass_reviveobj`, and `pass_sync` and two additional concepts: the *pnode number* and the *provenance record*.

A pnode number is a unique ID assigned to an object at creation time. It is a handle for the object's provenance, akin to an inode number, but never recycled. A provenance record is a structure containing a single unit of provenance: an attribute/value pair, where the attribute is an identifier and the value might be a plain value (integer, string, etc.) or a cross-reference to another object. Provenance records may contain *ancestry* information, records of data flows, or *identity* information.

The `pass_read` and `pass_write` operations are like `read` and `write` but are provenance-aware. This ensures that provenance and data move together, providing consistency of provenance and data as required by Section 4.

The `pass_read` call returns both the data requested and the exact identity of what was read: the file's *pnode* number and version as of the moment of the read. This ensures that applications or other higher layers can construct provenance records that accurately describe what they read, also a critical component of consistency.

The `pass_write` call takes both a data buffer and a "bundle" of provenance records that describe the data. A provenance bundle is an array of object handles and records, each potentially describing a different object. The complete provenance for a block of data written to a file might involve many objects (e.g., several processes

and pipes in a shell pipeline). This organization allows all of the separate objects to be sent as a single unit.

Cycle-breaking sometimes requires creating new versions of objects. In a layered system, versions must be handled at the bottom level (the storage system), but cycle-breaking may occur at any level. The `pass_freeze` call breaks cycles by requesting a new version.

As discussed in Section 4, provenance-aware applications may need to represent objects, such as browser sessions, data sets, or program variables, that do not map to a particular file system object. The `pass_mkobj` call allows applications to create such objects. These objects are referenced like files, with file handles. The objects can also be used to relate names/objects at one level to names/objects at another level. A system at any layer can create objects using `pass_mkobj` and create dependencies between its objects and objects at different layers of abstraction by issuing `pass_write` calls. Users can then issue queries using the name in the most convenient abstraction layer (e.g., filename) and PASSv2 can retrieve the appropriate objects across the layers using these dependencies.

We initially designed the objects returned by `pass_mkobj` to be transient and applications had no means to access these objects again after closing them. However, when developing provenance-aware applications, we discovered occasions where we needed to access these objects. Hence, we added the `pass_reviveobj` call that takes a pnode number and version and returns an object previously created via `pass_mkobj`.

By default, the provenance associated with an object returned via `pass_mkobj` is not flushed to disk unless it becomes a part of the ancestry of a persistent object on a PASS-enabled volume. This is correct behavior for purely transient objects with no descendants (e.g., processes), but it would lose objects that exist only at layers above PASS. Applications can use the `pass_sync` function to make persistent provenance associated with an object created via `pass_mkobj` even if is not in the ancestry of a persistent PASS-volume object.

Applications link against `libpass` to use the user-level DPAPI to record provenance. Such applications are *provenance-aware*. The DPAPI enables an arbitrary number of layers of provenance-aware applications. For example, we can construct a system with five layers using a provenance-aware Python application that uses a provenance-aware Python library, both of which execute on a provenance-aware Python interpreter. That provenance-aware Python interpreter might then use a PA-NFS utilizing PASSv2. Note that the provenance-aware library and provenance-aware interpreter both accept DPAPI calls from *higher* layers and issue DPAPI calls to *lower* layers.

5.3 Provenance Generation

Provenance generation involves two system components: the *interceptor* and *observer*. The interceptor captures system call events and reports them to the observer. The PASSv2 interceptor handles the following system calls: `execve`, `fork`, `exit`, `read`, `readv`, `write`, `writev`, `mmap`, `open`, and `pipe`, and the kernel operation `drop_inode`. The interceptor is a thin operating system specific layer, while the remaining system components can be mostly operating system independent.

The observer takes the information it receives from the interceptor, constructs provenance records, and passes those records to the analyzer via DPAPI calls. For example, when a process issues a `read` system call, the observer first issues a `pass_read` on the file. When the `pass_read` returns with the data, pnode, and version of the file, the observer creates a record stating that the particular version of the file is an input to the process, thereby creating a dependency between the process and the file. It then sends the record to the analyzer by issuing a `pass_write` with the provenance record, but no data. When that process then issues a `write` system call, the observer creates a record stating that the process is an input to the written file and issues a `pass_write` containing both this provenance record and the data from the `write` system call, thereby creating a dependency between the process and the file.

The observer is also the entry point for provenance-aware applications that use the DPAPI to explicitly disclose provenance records to PASSv2. The observer is the appropriate entry point, since PASS might need to generate additional provenance records even when an application is disclosing provenance. For example, when an application invokes a `pass_write` DPAPI call, apart from the explicit provenance disclosed by the application, the observer has to create a record that captures the dependency between the application and the file. The observer converts the provenance that higher-level provenance-aware applications explicitly disclose via DPAPI calls into appropriate kernel structures and passes the records to the analyzer.

5.4 Analyzer

The *analyzer* eliminates redundant provenance and cycles in the stream of provenance records that it receives. Programs generally perform I/O in relatively small blocks (e.g., 4 KB), issuing multiple reads and writes when manipulating large files. Each `read` or `write` call causes the observer to emit a new record, most of which are identical. The analyzer removes such

duplicates. Meanwhile, cycles can occur when multiple processes are concurrently reading and writing the same files. The analyzer prevents cycles by creating new versions of objects. In PASSv1, we used an algorithm that maintains a global graph of object dependencies and explicitly checks for cycles. On detecting a cycle, the algorithm merged all the nodes in the cycle into a single entity. This proved challenging, and there were cases where we were not able to do this correctly. In PASSv2, we use a more conservative algorithm, called the *cycle avoidance* algorithm that uses only an object's local dependency information to avoid cycles. We discuss and analyze this algorithm in detail in earlier work [20]. Since any semantic information that the higher-level applications disclose to PASSv2 is via objects returned through `pass_mkobj`, the analyzer works in a layered environment without modification.

5.5 Distributor

Since processes are first-class objects, the system must track and store their provenance. However, processes are not by themselves persistent objects residing on a PASS-enabled volume. Where should their provenance be stored? Similar issues arise with pipes, files from non-PASS volumes, and objects introduced by provenance-aware applications. In these cases, PASSv2 must select some PASS-enabled volume on which to store their provenance. The *distributor* addresses this issue.

The distributor caches provenance records for all objects that are not PASS files. When those objects become part of the ancestry of a persistent object on a PASS-enabled volume or are explicitly flushed via `pass_sync`, the distributor assigns these objects to a PASS volume (either that of the persistent ancestor or the one specified when an object was created) and flushes the provenance records by issuing a `pass_write` to Lasagna.

5.6 Lasagna & Waldo

Lasagna is our provenance-aware file system that stores both provenance and data. Lasagna is a stackable file system, based upon the eCryptfs [12] code base. Lasagna implements the DPAPI interface in addition to the regular VFS calls. We implement `pass_read`, `pass_write`, `pass_freeze` as inode operations and `pass_mkobj` and `pass_reviveobj` as superblock operations.

PASSv1 wrote provenance directly into databases that provided indexed access to provenance. This arrangement was neither flexible nor scalable, so PASSv2 writes all provenance records to a log. A user-level daemon process, `Waldo`, later moves the provenance to a database and indexes it. When the log file exceeds a parametrized maximum size or has been dormant for a parametrized length of time, the kernel closes the log and creates a new one. Waldo uses the Linux `inotify` interface to monitor this activity, processing and removing log files.

We use a write-ahead-provenance (WAP) protocol to ensure that on-disk provenance accurately reflects on-disk data. WAP is analogous to database write-ahead logging. Enforcing WAP requires that all provenance records be written to disk before the data they describe. This eliminates the possibility that unprovenanced data exists on the disk. In addition, we use transactional structures in the log along with MD5sums of data so that during file system recovery, we identify any data for which the provenance is inconsistent. This indicates precisely the data that was being written to disk at the time of a crash. Thus, Lasagna's DPAPI interface along with the WAP protocol ensures that provenance is consistent with the data it describes (or, after a crash, inconsistencies are identified).

5.7 Querying

Most existing provenance systems use either an XML-based or a relational representation. We found both lacking. XML has a notion of paths (XPath) but is inherently tree-structured and does not extend well to graphs. SQL has no native concept of paths; writing path-like queries in SQL requires mentally translating the paths into recursive queries, which are themselves expensive and unnatural in a relational environment. It seemed most appropriate to find a query language that was designed specifically for querying graphs.

The Lore semistructured database project at Stanford provided us with the Lorel [1] query language and its "OEM" data model. A semistructured database is one with no fixed schema; the data model in Lore is that of a collection of arbitrary objects, some holding values and some holding tables of named linkages to other objects. The data types of values and linkages are not fixed, and the query language is designed accordingly.

The OEM data model is appealing for provenance, since it naturally represents both graphs and object attributes, and Lorel provides the path-oriented query model for which we were looking. Unfortunately, we found that Lorel had several shortcomings. In particular, it did not support boolean values in the database, its formal grammar was ambiguous, and there were corner cases where the semantics were not well defined. We also needed to extend Lorel to allow traversal of graph edges in both directions. We present a more in-depth discussion of these issues in a recent publication [16].

We developed a new query language based on Lorel, which we call Path Query Language (PQL or "pickle"). It is specifically geared to handle our requirements for

querying provenance. PQL's query model is based on following paths through an object graph to find and retrieve data. The typical query returns a set of values. The general structure of a PQL query is: **select** *outputs* **from** *sources* **where** *condition*. Sources are path expressions, which represent paths through the graph, outputs are anything we can compute on paths, and conditions are boolean predicates like in a SQL query. The PQL reference manual is available online [15].

The following sample query determines the cause of the anomaly in the output in the use case described in Section 3.1.

```
select Ancestor
from Provenance.file as Atlas
     Atlas.input* as Ancestor
where Atlas.name = "atlas-x.gif"
```

The query returns all the ancestors of one output file, `atlas-x.gif` (by following zero or more `input` relationships), which will include both the Kepler workflow entities and the PASS data for the input files. PQL queries, if not posed carefully, can result in information overload. Pruning the query results to produce more focused results is an area of ongoing research.

6 Provenance-Aware Applications

The following sections present technical details about how we implemented provenance collection in a variety of different provenance-aware layers, and the provenance we collect in each. We conclude this section with a summary of the lessons learned while constructing these provenance-aware components. Table 1 summarizes the provenance collected by each provenance-aware system.

6.1 Provenance-Aware NFS

We implemented provenance-aware NFS using NFSv4 [26] in Linux 2.6.23.17. Making NFS provenance aware involves addressing two questions: First, while provenance-aware NFS can leverage the PASSv2 analyzer, should that analyzer reside on the client or the server? And second, how do we extend the NFSv4 protocol to support the six DPAPI operations?

6.1.1 Cycles vs. NFS

An analyzer must process all the provenance records at its abstraction layer in order to properly avoid cycles. Consider a process on an NFS client machine accessing data from two different storage servers. The analyzer must reside at the client, because only there is it possible to see all relevant provenance records.

Record Type	Description
PA-NFS	
`BEGINTXN`	Beginning record of a transaction
`ENDTXN`	Terminating record of a transaction
`FREEZE`	Freeze record sent in `pass_write`
PA-Kepler	
`TYPE`	Type of object: set to *OPERATOR*
`NAME`	Name of the operator
`PARAMS`	Operator parameters
`INPUT`	Dependency between operators
PA-links	
`TYPE`	Type of object: set to *SESSION*
`VISITED_URL`	Session and URL dependency
`FILE_URL`	File and URL dependency
`CURRENT_URL`	URL user was viewing while download was initiated
`INPUT`	File and Session dependency
PA-Python	
`TYPE`	Type of object: e.g., *FUNCTION*
`NAME`	object name (e.g., method name)
`INPUT`	method input and invocation dependency or invocation and output dependency

Table 1: Provenance records collected by each provenance-aware application.

Next, consider two programs running on different clients accessing a single server. By the same logic, the analyzer must reside on the server, because only there can it see all related provenance records.

Finally, combine these two scenarios: two client programs each accessing files from two different file servers. In this case, we need analyzers on both clients and servers.

This means that in general we must have an analyzer on every client and also an analyzer on every server; this in turn means that the client instance of the analyzer must be able to stack on top of the server instance, which means that the input and output data representations must be the same. This requirement is easily satisfied as all the components in PASSv2, including the client and the server, communicate via the DPAPI. In fact, it was precisely this observation that motivated layering and the use of the DPAPI as a universal interface.

6.1.2 DPAPI in NFS

pass_write: Supporting `pass_write` requires that we transmit provenance with data to enforce consistency. Accordingly, we created an NFS operation analogous to the local `pass_write`, called `OP_PASSWRITE`, that transmits both data and provenance to the server. As long as the combined data and provenance size is less than the

NFSv4 client's block size (typically 64 KB in NFSv4), this approach is sufficient.

Unfortunately, not all data and provenance packets satisfy this constraint. In these cases, we use NFS transactions to encapsulate a collection of operations that must be handled atomically by the server. To support transactions, we introduced two new operations, `OP_BEGINTXN` and `OP_PASSPROV`, and two new provenance record types, `BEGINTXN` and `ENDTXN`. First, we invoke an `OP_BEGINTXN` operation to obtain a transaction ID from the exported PASS volume. We record the transaction ID in a `BEGINTXN` record at the server. Then, we send the provenance records to the server in 64 KB chunks, using a series of `OP_PASSPROV` operations, each identified by the transaction ID acquired by `OP_BEGINTXN`. Finally, we invoke an `OP_PASSWRITE` operation that transmits the data along with a single `ENDTXN` record. The `ENDTXN` record contains the transaction ID obtained in `OP_BEGINTXN` and signals the end of that transaction. A corresponding `ENDTXN` record is written to the log at the server.

We considered an alternate implementation that obtains a mandatory lock on the file, writes the provenance, and then writes the data as a separate operation. This approach would have provided the coupling between provenance and data; however, it does not allow us to recover from a client crash. If the client wrote the provenance, crashed before sending the data, and then came back up, there is no way for the server to determine that the provenance must be discarded. Our implementation solves this problem, because the transaction ID enables the server's Waldo daemon to identify the orphaned provenance.

pass_read: For NFS `pass_read`, we introduced a new operation `OP_PASSREAD`, which returns both the requested data and its pnode number and version.

pass_freeze: We send `pass_freeze` operations to the server as a provenance record type in `OP_PASSWRITE`. When the analyzer at the client issues a `pass_freeze`, the client increments the version locally and attaches a freeze record to the file. The client can then return the correct version of the file on a `pass_read` without a trip to the server. Later, when the client sends the file's provenance to the server with an `OP_PASSWRITE`, the server processes the freeze records, incrementing its version number accordingly.

We implement freeze as a record type instead of an operation because operations may arrive at the server out of order. `pass_freeze` is order-sensitive with respect to `pass_write`. `pass_freeze` breaks cycles in the records that are about to be written with a `pass_write` and an out of order arrival can result in a failure to break cycles. Making `pass_freeze` a record type couples `pass_freeze` with `pass_write` and avoids the problem.

Due to the close-to-open consistency model that NFS supports, two different clients can open the same version of a file and concurrently make modifications to it. Hence, our approach of versioning at the client and updating versions at the server can lead to *version branching*, where two clients create independent copies of an object with the same version. This has not caused any problems in our existing applications, and given the overall lack of precise consistency semantics in NFS, we do not expect it to be problematic for existing applications.

pass_mkobj: We added a new operation called `OP_PASSMKOBJ` that returns a unique pnode referencing the object in future interactions. The client then constructs an in-memory anonymous inode that has a reference to the pnode and exports the inode to user-level as a file.

We could have implemented `pass_mkobj` by creating a file handle at the server and returning it to the client. The client would then use the handle to write provenance. However, this approach would make it difficult to recover from either a server or client crash. The advantage of our approach is that the server only needs enough state to verify that the pnode is a valid on `pass_reviveobj` and requires no complicated recovery. If the server crashes and comes back up, the client can continue to use the pnode as though the crash never happened, as the pnode is just a number. Similarly, if the client crashes, the server does not have to clean up state as it has only allocated a (cheap) pnode number to the client.

pass_reviveobj: We added a new operation called `OP_PASSREVIVEOBJ` that verifies that the given pnode number is valid and returns an anonymous inode as we do for `pass_mkobj`.

pass_sync: This is implemented by invoking the `OP_PASSPROV` operation. When the provenance exceeds 64KB, we encapsulate the operation in a transaction as we do for `pass_write`.

6.2 Provenance-Aware Kepler

Kepler records provenance for all communication between workflow operators, recording these events either in a text file or relational database. We added a third recording option: transmitting the provenance into PASSv2 via the DPAPI. This integration was simple. We implemented methods in Kepler's provenance recording interface that translate Kepler's provenance events into explicit ancestor-descendant relationships.

We create a PASS object for every workflow operator using `pass_mkobj` and set its properties, such as NAME, TYPE, and PARAMS, which specify the names and values of its parameters (such as "fileName" or "confirmOverwrite" for a file output operator). When an op-

erator produces a result, Kepler notifies our recording interface with its event mechanism. Upon receipt of the event, we add an ancestry relationship between this operator and every recipient of the message by issuing a `pass_write` call that records the ancestry between the sender and the recipient. This is the only one of Kepler's recording operations that needs to send data to PASSv2.

Unfortunately, the recording interface does not provide methods to generate provenance for reading or writing files or downloading data from the Internet. Instead, Kepler knows about data sink and source operators, which open and close files. We modified the Kepler routines used by these operators to infer the files that are being read/written, linking Kepler's provenance to that in PASSv2.

6.3 Provenance-Aware `links`

We chose to add provenance collection to version 0.98 of `links`, a text-based browser, as it had the simplest code base of those browsers we examined. We are currently exploring provenance collection in a Firefox [19].

A PA-browser can capture semantic information that is invisible to PASS, such as:

- The URL of any file that a user downloads using the browser;
- The web page a user was examining when she initiated a download;
- The sequence of web pages a user visited before downloading a file; and
- The set of pages that were active concurrently.

We group provenance by session, as it represents a logical task performed by a user. On session creation, we create a PASS object that represents it (using `pass_mkobj`) and record the object `TYPE` (using `pass_write`). Whenever a user visits a site, we generate a `VISITED_URL` record that describes the dependency between the session and the URL and record it by issuing the DPAPI call `pass_write`. These records identify the sequence of URLs that a user visited before downloading a file.

Each time the browser downloads a file, we generate three records. An `INPUT` record captures the dependency between the file and the session, connecting the file to the sequence of URLs visited during the session, before initiating the download. A `FILE_URL` record captures the URL of the file itself. A `CURRENT_URL` record captures the dependency between the file and the page the user was viewing when she decided to download the file. We replace the `write` that the browser issues to record the file on disk with a `pass_write` that transmits the data and the three provenance records to PASSv2.

6.4 Provenance-Aware Python Apps

We discovered that a colleague had written a set of wrappers to track provenance in Python applications. His goal was to explicitly identify relationships between input and output files using Python scripts that read in a large number of data files, but used only a subset of them.

To make the Python analysis program provenance-aware, we created Python bindings for our DPAPI interface. We also wrap objects, modules, basic types, and output files with code that creates PASSv2 objects representing our Python objects (using `pass_mkobj`), intercepts method invocations, and then records the relationships between the objects. By wrapping a few modules and objects we record the information flow pertaining to those objects and methods and relate them to the files they eventually affect. For every object, we record the object `TYPE` (for example, `FUNCTION`) and the object `NAME`. For modules and methods, we add an intercept for each method so we can connect method invocations to their inputs and outputs. On every method invocation, we issue DPAPI `pass_write` calls to record `INPUT` records describing the dependencies between each input and its method invocation and between the method invocation and each of its outputs.

6.5 Summary and Lessons Learned

While provenance-aware applications are generally useful and many developers develop *ad hoc* solutions to the problems they solve, the ability to integrate such solutions with system-level approaches increases the value of both the system-level provenance and the application-level provenance. Our system has a simple architecture and API that enables such an integration. We now discuss some of the lessons we learned.

Our experience with `links`, Kepler, and Python led us to the following guidelines for making applications provenance-aware. First, application developers have to identify the provenance they want to collect. Next, they have to replace read calls with `pass_read` calls and write calls with `pass_write` calls, obtaining and forwarding provenance to the layers around them. In order to record semantic provenance, application developers can create objects using `pass_mkobj` and record such provenance via `pass_write` calls on those objects. They can then link the semantic provenance with the system objects by creating appropriate records and storing them via `pass_write` calls. Finally, layers that are a substrate to higher level applications (like an interpreter) must export the DPAPI. If they do not export the DPAPI, the applications cannot layer provenance on top of them.

It is not trivial to extend existing complex applications,

which were not designed to collect provenance, to make them provenance aware. We observe this in our ongoing work with Firefox. In Firefox, interesting provenance events such as page loads, bookmarks, etc. occur in the user interface modules. However, the I/O manipulation events such as cache and file writes occur in completely different modules. Connecting provenance collected in the UI modules to data writes in the I/O modules entails a significant amount of re-engineering of Firefox modules and interfaces. We are currently working on this.

Considering that operating systems are, to this day, introducing new system calls, we expect the DPAPI to evolve over time. It has continued to evolve over the course of the project and this paper. As we noted in Section 5.2, we initially designed the objects returned by `pass_mkobj` to be transient. However, while working on Firefox provenance collection, we discovered that we needed to revive these objects. For example, in Firefox, we create an object per active session. Firefox stores the sessions to disk and restores them when the user restarts the browser. In this scenario, the application needs to revive the objects used to record each session's provenance so as to record further provenance. Hence, in order to support such scenarios, we extended the DPAPI to include `pass_reviveobj`.

We initially believed that the Python wrappers we built were sufficient to enable provenance-aware Python applications. We later realized that while we could wrap functions, we lost provenance across built-in operators. In retrospect, what we discovered with Python was the difference between building a provenance-aware system and provenance-aware applications. By wrapping function calls in Python, we make an application provenance-aware, as we did for `links` and Kepler. Making Python itself provenance-aware would require modifying the Python interpreter, as we modified the operating system to make it provenance-aware. While an interesting project, we have left that undertaking for future research.

7 Performance Evaluation

While the main contribution of this work is in the new capabilities available from the system, we wanted to verify that these capabilities do not impose excessive overheads. There are two concerns: the execution time overhead due to the additional work done to collect provenance and the space overhead for storing provenance.

We evaluate these overheads using five applications representative of a broad range of workloads: 1) Linux compile, in which we unpack and build Linux kernel version 2.6.19.1. This represents a CPU intensive workload; 2) Postmark, that simulates the operation of an email server. We ran 1500 transactions with file sizes ranging from 4 KB to 1 MB, with 10 subdirectories and 1500 files. This benchmark is representative of an I/O intensive workload; 3) Mercurial activity benchmark, where we evaluate the overhead a user experiences in a normal development scenario. We start with a vanilla Linux 2.6.19.1 kernel and apply, as patches, each of the changes that we committed to our own Mercurial-managed source tree; 4) Blast, a biological workload used to find protein sequences in a species that are closely related to the protein sequences in another species. The workload formats two input data files with a tool called `formatdb`, then processes the two files with Blast, and then massages the output data with a series of Perl scripts; and 5) A PA-Kepler workload, that parses tabular data, extracts values, and reformats it using a user-specified expression. The PA-Kepler workload, when located on a PA-NFS volume, is similar to the situation presented in Section 1, where provenance collection is integrated across three layers.

We ran two batches of experiments: one comparing PASSv2 to vanilla ext3 (in ordered mode) and another on comparing provenance-aware NFS (PA-NFS) to NFS exporting ext3 (also in ordered mode). We ran all local benchmarks on a 3GHz Pentium 4 machine with 512MB of RAM, an 80GB 7200 RPM Western Digital Caviar WD800JB hard drive and with a kernel (Vanilla/PASS) based on Linux 2.6.23.17. For experiments involving NFS, we used the previous machine as the server and a 2.8GHz 2 CPU Opteron 254 machine with 3GB of RAM as the client. The client machine has the same software configuration as the server.

PASSv2 Elapsed Time Results: Table 2 shows the elapsed time overheads. The general pattern we observed is that the elapsed times are affected when provenance writes interfere with the workload's regular writes. The Mercurial activity benchmark has the highest elapsed time overhead of 23.1% despite having minimal space overhead. This is because `patch` performs many metadata operations (it creates a temporary file, merges data from the patch file and the original file into the temporary file, and finally renames the temporary file). The provenance writes interfere with `patch`'s metadata I/O, leading to extra seeks, which increase overhead. The Linux kernel compile has an overhead of 15.6% due to provenance writes. Postmark has an overhead of 11.5%, and the overheads, unlike the former two benchmarks, are due to the double buffering in Lasagna (stackable file systems cache both their data pages and lower file system data pages). Blast and PA-Kepler are heavily CPU bound and hence their elapsed time are minimally affected due to provenance writes.

PA-NFS Elapsed Time Results: The PA-NFS elapsed time overheads are lower for Linux compile and Mercurial activity benchmarks compared to PASSv2 overheads, as the additional delay introduced by the net-

Benchmark	Ext3	PASSv2	Overhead	NFS	PA-NFS	Overhead
Linux Compile	1746	2018	15.6%	3320	3353	11.0%
Postmark	453	505	11.5%	636	743	16.8%
Mercurial Activity	614	756	23.1%	2842	3089	8.7%
Blast	69	69.5	0.7%	52	53	1.9%
PA-Kepler	1246	1264	1.4%	160	164	2.5%

Table 2: Elapsed time overheads (in seconds).

Benchmark	Ext3	Provenance	Provenance+Indexes
Linux Compile	1287.9	88.9 (6.9%)	236.8 (18.4%)
Postmark	1289.5	0.8 (0.1%)	1.7 (0.1%)
Mercurial Activity	858.7	15.4 (1.8%)	28.9 (3.4%)
Blast	5.6	0.1 (1.1%)	0.2 (3.8%)
PA-Kepler	3.5	0.2 (4.7%)	0.5 (14.2%)

Table 3: Space overheads (in MB) for PASSv2. The space overheads for PA-NFS are similar.

work round trips affect both NFS and PA-NFS equally. The Postmark overheads, though reasonable, are higher in PA-NFS compared to PASSv2. Our experiments confirm that out of 16.8% overhead that Postmark incurs for PA-NFS, 14.8% is due to the fact that Lasagna is implemented as a stackable file system. The Blast and PA-Kepler overheads remain minimal even in the PA-NFS case.

Space Overheads: Table 3 shows the provenance database space overhead and the total space overheads (provenance database and indexes) for PASSv2. The overheads are computed as a percentage of the Ext3 space utilization. Overall, the provenance database overheads are minimal with all overheads being less than 7%. The total space overheads are reasonable with Linux compile having the highest overhead at 18.4%. PA-Kepler combines both system provenance and application provenance and has a total space overhead of 14.2%. For the rest of the benchmarks, the space overhead is less than 4%. The space overheads for PA-NFS as the overheads are similar to the overheads in PASSv2.

8 Related Work

Several systems have looked at propagating taint information along with the data in order to debug applications or to detect security violations [22, 30]. These systems are, however, extremely slow as they track information flow at a fine granularity and hence can never be used in production systems. PASSv2 monitors only system call events, which is much less expensive. The drawback is that the information collected by PASSv2 can be less accurate; but as we have shown in the use cases, it is valuable nonetheless.

X-Trace [7] is a research prototype built to diagnose problems in network applications that span multiple protocol layers and administrative domains. X-Trace's approach is similar to ours in that it integrates information from multiple layers in the system stack. The higher layer generates a "taint" that is propagated through the layers along with the data. The generated debug metadata is not sent with the data, but is instead sent out of band to a destination. Hence the interface between layers can be much more limited compared to the DPAPI in PASSv2. Furthermore, X-Trace does not need to deal with cycles as the PASSv2 analyzer does.

Another class of systems that maintain dependencies are software build systems such as Vesta [14]. These systems need the initial dependencies be specified manually. Build systems maintain dependencies after those dependencies have been specified; PASS derives dependencies based upon program execution. As a result, while extraordinarily useful for software development, they ignore the central PASS challenge: automatically generating the derivation rules as a system runs.

Chanda et. al. [5] present a mechanism to use causal information flow to introduce new functionality. For example, one can send process priority along with data to a socket. On receiving the data and the causal metadata (priority), the server increases the priority for processing this data. This mechanism is complementary to the ideas we have explored in this work.

9 Conclusions

We have presented a provenance-aware storage system that permits integration of provenance across multiple layers of abstraction, ranging from Python applications to network-attached storage. This integration requires a layered architecture that dictates how provenance, data,

and versions must flow through the system. The architecture has proved versatile enough to facilitate integration with a variety of applications and NFS, providing functionality not available in systems that cannot integrate provenance across different layers of abstraction.

We have presented several use cases illustrating what kinds of functionality this layering enables. The use cases show efficacy in a variety of areas, such as malware tracking and scientific data processing. Finally, we demonstrated an end-to-end system encompassing provenance-aware applications and network-attached storage, imposing reasonable space and time overheads ranging between 1% and 23%.

Acknowledgments: We thank Stephen D. Holland for providing us the PA-Python use cases and Joseph Barillari for help with PA-Python development. We thank Andrew Warfield, our shepherd, for repeated careful and thoughtful reviews of our paper. We thank Keith Bostic, Stephen D. Holland, Keith Smith, and Jonathan Ledlie for their feedback on early drafts of the paper. We thank the anonymous reviewers for the valuable feedback they provided. This work was partially made possible thanks to NSF grants CNS-0614784 and IIS-0849392.

References

[1] ABITEBOUL, S., QUASS, D., MCHUGH, J., WIDOM, J., AND WIENER, J. L. The Lorel query language for semistructured data. *International Journal on Digital Libraries 1*, 1 (1997), 68–88.

[2] ALTINTAS, I., BARNEY, O., AND JAEGER-FRANK, E. Provenance collection support in the Kepler scientific workflow system. In *IPAW* (2006), vol. 4145 of *LNCS*, Springer.

[3] BOSE, R., AND FREW, J. Composing lineage metadata with xml for custom satellite-derived data products. In *Proceedings of the Sixteenth International Conference on Scientific and Statistical Database Management* (2004).

[4] BRAUN, U., SHINNAR, A., AND SELTZER, M. Securing Provenance. In *Proceedings of HotSec 2008* (July 2008).

[5] CHANDA, A., ELMELEEGY, K., COX, A. L., AND ZWAENEPOEL, W. Causeway: operating system support for controlling and analyzing the execution of distributed programs. In *HOTOS* (2005).

[6] EFSTATHOPOULOS, P., KROHN, M., VANDEBOGART, S., FREY, C., ZIEGLER, D., KOHLER, E., MAZIÈRES, D., KAASHOEK, F., AND MORRIS, R. Labels and event processes in the asbestos operating system. In *SOSP* (2005).

[7] FONSECA, R., PORTER, G., KATZ, R. H., SHENKER, S., AND STOICA, I. X-trace: A pervasive network tracing framework. In *In NSDI* (2007).

[8] FOSTER, I., AND KESSELMAN, C. Globus: A metacomputing infrastructure toolkit. *International Journal of Supercomputer Applications and High Performance Computing* (Summer 1997).

[9] FOSTER, I., VOECKLER, J., WILDE, M., AND ZHAO, Y. The Virtual Data Grid: A New Model and Architecture for Data-Intensive Collaboration. In *CIDR* (Asilomar, CA, Jan. 2003).

[10] GenePattern. `http://www.broad.mit.edu/cancer/software/genepattern`.

[11] GOEL, A., PO, K., FARHADI, K., LI, Z., AND DE LARA, E. The Taser intrusion recovery system. In *SOSP* (2005).

[12] HALCROW, M. A. eCryptfs: An enterprise-class encrypted filesystem for linux. *Ottawa Linux Symposium* (2005).

[13] HASAN, R., SION, R., AND WINSLETT, M. The Case of the Fake Picasso: Preventing History Forgery with Secure Provenance. In *FAST* (2009).

[14] HEYDON, A., LEVIN, R., MANN, T., AND YU, Y. The Vesta Approach to Software Configuration Management. Technical Report 168, Compaq Systems Research Center, March 2001.

[15] HOLLAND, D. A. PQL language guide and reference. `http://www.eecs.harvard.edu/syrah/pql/docs/`. Harvard University, 2009.

[16] HOLLAND, D. A., BRAUN, U., MACLEAN, D., MUNISWAMY-REDDY, K.-K., AND SELTZER, M. I. A Data Model and Query Language Suitable for Provenance. In *Proceedings of the 2008 International Provenance and Annotation Workshop (IPAW)*.

[17] KING, S. T., AND CHEN, P. M. Backtracking Intrusions. In *SOSP* (Bolton Landing, NY, October 2003).

[18] KROHN, M., YIP, A., BRODSKY, M., CLIFFER, N., KAASHOEK, M. F., KOHLER, E., AND MORRIS, R. Information flow control for standard OS abstractions. In *symposium on Operating systems principles* (2007).

[19] MARGO, D. W., AND SELTZER, M. The case for browser provenance. In *1st Workshop on the Theory and Practice of Provenance* (2009).

[20] MUNISWAMY-REDDY, K.-K., AND HOLLAND, D. A. Causality-Based Versioning. In *Proceedings of the 7th USENIX Conference on File and Storage Technologies* (Feb 2009).

[21] MUNISWAMY-REDDY, K.-K., HOLLAND, D. A., BRAUN, U., AND SELTZER, M. Provenance-aware storage systems. In *Proceedings of the 2006 USENIX Annual Technical Conference*.

[22] NEWSOME, J., AND SONG, D. X. Dynamic taint analysis for automatic detection, analysis, and signature generation of exploits on commodity software. In *NDSS* (2005).

[23] Provenance aware service oriented architecture. `http://twiki.pasoa.ecs.soton.ac.uk/bin/view/PASOA/WebHome`.

[24] The First Provenance Challenge. `http://twiki.ipaw.info/bin/view/Challenge/FirstProvenanceChallenge`.

[25] The Second Provenance Challenge. `http://twiki.ipaw.info/bin/view/Challenge/SecondProvenanceChallenge`.

[26] SHEPLER, S., CALLAGHAN, B., ROBINSON, D., THURLOW, R., BEAME, C., EISLER, M., AND NOVECK, D. Network File System (NFS) version 4 Protocol. `http://www.ietf.org/rfc/rfc3530.txt`, April 2003.

[27] SUNDARARAMAN, S., SIVATHANU, G., AND ZADOK, E. Selective versioning in a secure disk system. In *Proceedings of the 17th USENIX Security Symposium* (July-August 2008).

[28] VAHDAT, A., AND ANDERSON, T. Transparent result caching. In *ATEC '98: Proceedings of the annual conference on USENIX Annual Technical Conference* (1998).

[29] WIDOM, J. Trio: A System for Integrated Management of Data, Accuracy, and Lineage. In *CIDR* (Asilomar, CA, January 2005).

[30] YIN, H., SONG, D., EGELE, M., KRUEGEL, C., AND KIRDA, E. Panorama: capturing system-wide information flow for malware detection and analysis. In *CCS* (2007).

[31] ZELDOVICH, N., BOYD-WICKIZER, S., KOHLER, E., AND MAZIÈRES, D. Making information flow explicit in histar. In *OSDI* (2006).